Data manipulation API in ERP systems

Vadym Borovskiy, Wolfgang Koch, and Alexander Zeier
Hasso Plattner Institute for Software Systems Engineering
P.O. Box 900460, D-14440 Potsdam, Germany
{vadym.borovskiy, alexander.zeier}@hpi.uni-potsdam.de, wolfgang.koch@sap.com

Abstract

Efficiency in data manipulation is of vital importance to ERP systems. Flexible data manipulation API helps to address a number of acute needs of end users and application developers. To be efficient the API must meet a number of requirements: (i) allow data operations at any granularity level, (ii) guarantee data integrity, (iii) be scalable, (iv) allow cross-platform invocations, (v) be convenient to use. None of the ERP systems exposes an API like that.

Building on the notions of business objects and Web services, this paper contributes with the concept of a business object query language (BOQL). Essentially, BOQL is a query-like service invocation, which provides on-the-fly orchestration of CRUD-operations of business objects. BOQL allows to achieve the desired level of data manipulation efficiency at a reasonable price. To demonstrate the efficiency of the suggested approach the paper presents a use case showing how BOQL enables the development of composite applications.

1 Introduction

Data manipulation is critical to enterprise resource planning (ERP) systems. An ERP system is a data-driven application, meaning that accessing and manipulating data represent the biggest share of its workload. The more efficient these operations, the greater the value of the system. The current paper elaborates on two questions: what makes ERP data manipulation efficient and how the efficiency can be achieved. The main contribution of the paper is the concept of a business object query language (BOQL). Essentially, BOQL is a query-like service invocation, which provides on-the-fly orchestration of CRUD-operations of business objects. BOQL allows to achieve the desired level of data manipulation efficiency at a reasonable price.

The rest of the paper is structured as follows. Section 2 discusses the requirements that an efficient data API must satisfy. Section 3 gives an overview of related work and analyzes the state of the art in ERP data access. Section 4 presents the business object query language (BOQL), which is the main contribution of the paper, and shows how it satisfies the requirements discussed in Section 2. Section 5 presents a prototype demonstrating how an ERP system can expose BOQL as an interface to its data. Section 6 presents a use case, where BOQL is used to develop a composite application. Section 7 concludes the paper.

2 Accessing and manipulating ERP data

To be efficient, data API in an ERP system must satisfy a number of requirements: (i) allow data operations at any granularity level, (ii) guarantee data integrity, (iii) be scalable, (iv) allow cross-platform invocations, (v) be convenient to use. Each requirement is explained below.

The first requirement, interprets efficiency as the ability to access any combination of attributes with as little effort as possible. An ERP system stores in its database hundreds of gigabytes of data. Typically, these data are fragmented. For example, a sales order header is stored separately from sales order items. The fragmentation is the direct consequence of data normalization, the process of ensuring the maximum cohesion of entity types and eliminating redundancy in data model. The main argument in favor of normalization is that the management of normalized data is simpler than that of denormalized. The disadvantage of normalization is that it disjoints semantically coupled attributes into a number of subsets and stores them separately (in different database tables). Data API in such systems (e.g. SAP R/3 and SAP Business Suite) is often organized around these subsets, meaning that the system exposes operations that are able to manipulate only the attributes that belong to a single table. This is very inefficient, because the semantics of data operations in most applications assumes manipulation on application domain objects (purchase order, customer invoice and etc.) rather than on database tables. In this case the API’s granularity does not match the granularity of the applications’ needs. Whenever such a mismatch happens, application developers have to take the burden of express-
ing the required operation in terms of available ones. This complicates the code and leads to situations, when data manipulation part of an application outweighs the application’s value adding part.

The second requirement, data integrity, ensures that any usage of the API must not compromise the integrity of ERP data. Data integrity means that the manipulation of data must be performed in accordance with business logic rules. For example, when creating a sales order, availability check must be performed. Only after a successful confirmation a sales order entry can be created in the system. Without such a check, creating a sales order by simply inserting data into corresponding database tables with SQL statements can result in a sale of not existing products.

The third requirement, scalability, ensures the ability of the target API to handle increasing number of requests. One of the current trends in ERP systems development is switching to software as a service (SaaS) model. No maintenance and a better pricing model based on actual usage drive organizations to favor SaaS ERP solutions over traditional on-premise ones [7]. Because the service provider rather than the consumer has to bear the cost of operating the system, the provider is strongly motivated to reduce the system’s total cost of ownership. A well-designed hosted service reduces total cost of ownership by leveraging economy of scale. Sharing all aspects of the IT infrastructure among a number of consumers allows the provider to reduce operating expenses and thus increase profit. Consolidation of consumers onto a singe operational system, however, comes at the price of increased system complexity due to multi-tenancy at the database layer. Multi-tenancy is basically mapping a number of single-tenant logical schemas in one multi-tenant physical schema in the database [4]. The inevitable consequence of multi-tenancy is increased workload: the system has to handle requests from much more consumers in comparison to a single-tenant system [8]. Therefore, scalability becomes a crucial characteristic of an ERP system, if its provider plans to offer it as a service.

The forth requirement, cross-platform support, is necessary in the contemporary heterogeneous world. The variety of platforms, from which applications access ERP systems, has dramatically increased over the last decade. Desktop computers running different operating systems, mobile devices, enterprise-class servers, all may run applications interacting with ERP systems. Designing specific API for every platform configuration is costly in terms of development and maintenance. Therefore, to support such a heterogeneous environment, the target API must be based on industry standards and avoid the usage of proprietary technologies. Standards compliance, however, must not be abused. It is a trade-off: by improving interoperability designers often sacrifice performance and simplicity of code.

The last requirement is the ease of use. Although obvious, the requirement often gets overlooked. As consequence, powerful functionality can become not usable. In the context of ERP data manipulation the main usability factor in our opinion is the transparency of data model. Therefore, the target API must not only expose data of an ERP system, but also the system’s data model or metadata.

3 Related work

A straightforward approach to data manipulation can be to use SQL. Since an ERP system relies on relational database, SQL statements could be issued directly against the database to operate on data. Although feasible, this approach is unlikely to satisfy the requirements stated above. The problem with SQL is that it violates the data encapsulation principle. It exposes too much control over the underlying database and increases the risk of corrupting data in the system. An ERP system is not only data, but also a set of business rules that apply to the data. Generally, these rules are not a part of the system’s database. Direct access to the database circumvents the rules and violates data integrity. Therefore, accessing directly the data by any means must be prohibited.

An alternative to SQL can be data as a service approach. In this case a system exposes a number of Web services with strongly-typed interfaces operating on data. This approach has an advantage of hiding internal organization of data. Instead of a data schema and a query interface an ERP system exposes a set of operations that manipulate its data. By choosing operations and calling them in an appropriate sequence required actions can be performed. Because of using Web services this approach is platform independent. In fact, the data-as-a-service approach has been very popular. SAP, for instance, has defined hundreds of Web service operations that access data in SAP Business Suite [2]. Amazon Electronic Commerce service is another example of such approach [1]. However, this method has a serious disadvantage - lack of flexibility. Although an ERP system can expose many data manipulation operations, they will unlikely cover all combinations of attributes that applications might need to operate on. Therefore, granularity mismatches are very likely to occur. As discussed earlier, this will require application developers to manually construct a sequence of calls on existing operations to perform a desired manipulation. An example of such a case is presented in [6]. To partially overcome the mismatch, the interfaces of Web services can be relaxed [5]. This, however, will blur the semantics of the operations.

Service data objects (SDO) [9] enhance the data-as-a-service approach by rigidly specifying many aspects of data manipulation API. SDO is a specification for a programming model that unifies data programming across hetero-
geneous data sources, provides robust support for common application patterns, and enables applications, tools, and frameworks to more easily query, view, bind, update, and introspect data [3]. SDO has a composable (as opposed to monolithic) architecture and is based upon the concept of disconnected data graphs. Under the disconnected data graphs pattern, a client application retrieves a data graph from a data source, mutates the it, and can then apply the data graph changes back to the data source. Access to data sources is provided by a class of components called data access services. A data access service (DAS) is responsible for querying data sources, creating graphs of data containing data objects, and applying changes to data graphs back to data sources. SDO essentially wrap data sources and fully control access to them via a set of strongly-typed or dynamic interfaces. SDO offer a number of advantages: data encapsulation, better semantics in comparison to the previous approach (because manipulation API is organized around data objects from application domain), better modularity and reuse. However, SDO have a weakness (as in the case of data-as-a-service approach): the problem of interface design is not solved. Therefore, granularity mismatches are possible, but the usage of dynamic interfaces can alleviate the problem at the cost of code complexity.

As one can see all approaches have advantages and disadvantages. SQL as a data access API gives great flexibility by allowing to construct queries that match the granularity of any information need. However, SQL exposes too much control over the database and circumvents business logic rules. The data-as-a-service and SDO approaches, on the other hand, enforce business rules by exposing a set of Web operations, which encapsulate data manipulation and hide data organization. However, the granularity of the exposed operations often does not match the needs of application developers due to poor interface design.

4 Business object query language

Forming a business object out of semantically related data and accessing the data via a fixed set of operations give good control over the data. On the other hand, exposing the data via a specific interface limits application developers in ways they can manipulate the data. To overcome this deficiency we suggest to use only a dynamic interface of a business object and automatically generate the sequence of calls to its operations, given a special query (compact, formal description of an action a developer wants to perform on the business object).

Despite the diverse semantics of business objects they all have the same structure (an array of attributes and associations) and behavior (a set of operations). The most basic set of operations a business object supports is called CRUD - Create, Retrieve, Update, Delete. Although too generic, this set of operations has an advantage that any business object can support it. Therefore, all business objects can be derived from the same base class featuring the mentioned arrays (of attributes and associations) and CRUD-operations. Such uniform behavior and structure allow to introduce a query language for business objects very much like SQL for relational entities. We propose the following scenario:

1. A programmer composes a query, the description of what to retrieve from or change in the system, according to some SQL-like grammar, and sends the query as a string to the system by calling a generic Web service operation, for example ExecuteQuery.

2. The system parses the string to detect present clauses (from, select, where, etc.) and builds a query tree - an internal representation of the query. The tree is then passed for further processing to a query execution runtime, very much like in a DBMS.

3. Using the from clause the runtime obtains references to the business objects, on which the operations must be performed - source business objects. Then the runtime traverses the query tree in a specific order and converts recognized query tokens to appropriate CRUD-calls on the source business objects. For example, tokens from select clause are converted to Retrieve or Retrieve-ByAssociationChain operations, while tokens from update clause are converted to Update operations.

4. Having constructed the call sequence, the runtime binds corresponding string tokens to the input parameters of CRUD-operations. For example, the token Customer.Name of the select clause is interpreted as a call to Retrieve operation with the input parameter value "Name" on the business object Customer. Now everything is ready to perform the calls of CRUD-operations in the on-the-fly constructed sequence.

5. The last step is the composition of result set. The result is compiled in an XML document and sent back to the calling application.

In its essence the BOQL performs on-the-fly orchestration of calls to objects’ CRUD-operations based on user-defined queries. These queries are transformed to a sequence of calls that perform the required action. BOQL has an advantage of supporting queries at any granularity level as in the case of SQL (because of using dynamic interface of business objects) without circumventing business rules (because CRUD-operations control data manipulation and enforce business logic rules). BOQL is made feasible by a uniform representation of business objects (in terms of the structure and behavior). As one can see BOQL already fulfills the requirements i, ii and iv from Section 2. The rest
Implementing BOQL

The current section demonstrates how an ERP system can support BOQL. The Figure 1 sketches the architecture of a prototyped system. BOQL is implemented by two elements: a business object engine and a query engine: the former manages business objects in a way BOQL assumes and the latter provides access to them from outside of the owning process via a query-like interface. These two elements are instances of BoEngine and QueryEngine classes respectively. Both are created at the system’s startup time. Business object engine is instantiated first to assemble business objects and store references to them in a pool. Then the instance of the query engine is created. It has access to the pool and can manipulate the objects.

Every business object encapsulates an in-memory table to cache data. The in-memory table is populated with data taken from a private database. Every object also encapsulates logic to synchronize its in-memory table with the database. To the query execution runtime an object is available via its interface: a collection of attributes and associations to other objects and CRUD-operations. How those are implemented is completely hidden inside the object. Neither business objects nor their in-memory cache and database tables can be directly accessed outside of the owning process. The direct access to the data is prohibited to enforce internal business logic. To access the business data an external application must use the standardized query-like interface exposed by the query engine. When the latter receives a query it parses it and transforms recognized tokens to corresponding operation invocations.

The implementation platform for the prototype is .NET. The system is implemented as a Windows service and the query interface is published as a Web service hosted by Internet Microsoft Information Services (IIS). The Web service is meant to dispatch a query to the system and serves as a request entry point. There is no other way to invoke or access the system except for issuing a call to the Web service. The physical data storage is implemented as a Microsoft SQL Server 2005 database.

To make the architecture scalable, it must be multithread-aware and bottle-neck-free. The Web service can easily become a bottleneck in the system. Therefore, the actual BOQL query execution is factored out from the Web service to so called working processes. Each process can run on any physical server, to which the Web service hosting machine can establish a TCP connection. Every working process has a queue, to which the Web service appends BOQL queries, and a pool of working threads, which pull out the queries from the queue and process them. The queue is very important, as it enables asynchronous communication between the Web service and the working process. The former does not wait for the completion of the query and becomes quickly available to other client applications. To determine a working process that should be assigned a BOQL query the Web service uses round-robin scheduling algorithm. Hence, if the number of queries increases, the system’s administrators must simply instantiate more working processes and register them with the Web service.

Using BOQL requires the knowledge of business object model: the list of business objects the system has, their attributes and associations. To communicate this information we have developed a tool called Schema Explorer. It automatically retrieves metadata from the system and builds a business object graph (see Figure 2). Such a tool greatly simplifies the creation of BOQL queries by offering a plenty of useful functionality: business object search, association and attribute search, finding connections/paths between any two business objects, displaying a business object graph or its part, intellisense support for query editor, test execution of a query, to name just a few. The business object metadata is obtained using reflection mechanism of .Net Framework. Query engine exposes a number operations which internally use reflection to query the business object metadata. For example, if a developer wants to know what business objects the system has, the tool calls an operation which scans the pool and obtains the types of business objects instantiated by the system. To look up the list of associations of a given business object, say Customer, the tool issues a call to another operation that gets the names of elements in the As-

---

1 meaning that it guarantees the compliance to CRUD-interface

2 of course free hardware capacity must be available
Figure 2. Schema of the Web retailer’s CRM

associations array of the corresponding business object. Because business object model does not change the metadata is cached in order to avoid the reflection overhead.

6 Example

Consider a Web retailer that sells items online and subcontracts a logistics provider to ship sold products. The retailer operates in a geographically large market (e.g. the US or Europe). In this situation the consolidated shipment of items can generate considerable savings in delivery. Consolidation means that a number of items is grouped in a single bulk and sent as one shipment. The savings come from price discounts gained from higher transportation volume and less transaction cost per item shipped (because it is the bulk that is charged, but not individual items).

To manage their sales, the retailer uses a system with the business object graph as Figure 2 presents. We assume that the system exposes a query-like Web service interface as described in the section 5. The query returning the shipping address for all sales order items that are to be delivered looks as follows:

```sql
SELECT
  SO.id, SO.Contact.Address.postalAddr,
  SO.Items.id, SO.Contact.Customer.name,
FROM
SalesOrder As SO
WHERE
SO.Status = "ToDeliver"
GROUP BY
  SO.Contact.Address.city
```

By invoking the query-liked Web service and passing the above query to it, a third-party application consolidates the items by their destination. The next step for the application is to submit a request for quote to a logistics provider and get the price of transporting each group of items. Many logistics providers have a dedicated service interface for this, so the application can complete this step automatically.

7 Conclusion

Efficient data manipulation API is essential to ERP systems. To be efficient, the API must satisfy a number of requirements: (i) allow data operations at any granularity level, (ii) guarantee data integrity, (iii) be scalable, (iv) allow cross-platform invocations, (v) be convenient to use. Existing approaches and APIs do not satisfy all these requirements. The current work contributes with the concept of query-like service invocation implemented in the form of a business object query language. BOQL offers both the flexibility of SQL and encapsulation of data-as-a-service and SDO approaches. In its essence, BOQL is on-the-fly orchestration of CRUD-operations exposed by business objects of an ERP system. The authors also showed that BOQL satisfies all five requirements. In addition the paper demonstrated how BOQL enables the development of enterprise composite applications. Furthermore, the major components of the architecture were outlined and prototyped with Microsoft .NET platform.

References